Modulo II
Spring Framework
AOP

Prof. Ismael H F Santos

Bibliografia

- Spring in Action
  - Craig Walls and Ryan Breidenbach
- Professional Java Development with Spring
  - Rod Johnson, Juergen Hoeller and Team
- Spring – www.springframework.org
- J2EE without EJB – Rod Johnson/ Jurgen Hoeller
- Better, Faster, Lighter Java – Bruce Tate
- Wiring your Web Application with Open Source Java http://www.onjava.com/pub/a/onjava/2004/04/07/wiringwebapps.html

Spring Related Tools and Add-Ons

- ACEGI Security - comprehensive security services for the Spring Framework
- Spring IDE - graphical user interface for the configuration files used by the Spring Framework
- Spring BeanDoc - tool that facilitates documentation and graphing of Spring bean factories and application context files
- XDoclet Spring Tags - support for generating Spring XML config files from annotations in Java classes (you could also use JDK1.5 annotations to achieve this)
- Spring Web Flow - for web applications with demanding page flow requirements
- AppFuse Not really a tool or add-on, but AppFuse is Matt Raible's project to jumpstart your Java web projects. It uses Spring at it's core and studying it is a great way to learn about Spring.
- Spring Framework .NET – Spring Clone for the Dark Side ☺

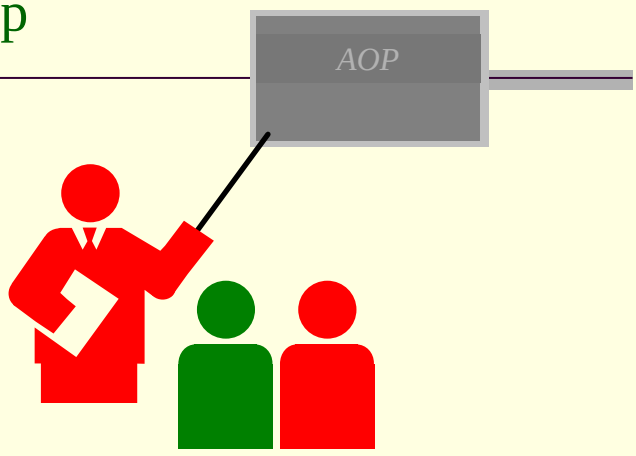
Spring Framework / Spring Related References

- **The Official Spring Reference Manual**
  http://www.springframework.org/docs/reference/
- **Introduction to Spring by Rod Johnson**
- http://www.theserverside.com/articles/article.tss?l=SpringFramework
- **Spring in Action** by Craig Walls and Ryan Breidenbach
- **Pro Spring** by Rob Harrop and Jan Machacek
- **J2EE Without EJB** by Rod Johnson and Juergen Holler
- **Expert One-on-One J2EE Design and Development** by Rod Johnson
- **Spring Developers Notebook** by Bruce Tate and Justin Gehtland
- **Better, Faster, Lighter Java** by Bruce Tate and Justin Gehtland
- **Spring Live** by Matt Raible
- **Professional Java Development with the Spring Framework**
- by many of the core Spring developers: **Coming in July 2005**

Ementa

- Spring AOP

WebApp



Spring AOP

- ***Three pillars of Spring: IoC, Consistency of Service Abstraction, AOP.***
- ***What is AOP***
  - *"Aspect-Oriented Programming (AOP) complements OOP by providing another way of thinking about program structure. While OO decomposes applications into a hierarchy of objects, AOP decomposes programs into aspects or concerns. This enables modularization of concerns such as transaction management that would otherwise cut across multiple objects. (Such concerns are often termed crosscutting concerns.)"[3]*

Conceitos de AOP

- Aplicações possuem códigos diversos como:
  - Gerenciamento de transações
  - Logging
  - Segurança
- Essas responsabilidades devem pertencer a implementação das classes?
  - A classe `AccountService` deve ser ser responsável pelo gerenciamento de transação, logging e segurança?

AOP (Aspect-Oriented Programming)

- AOP enables the delivery of services to POJOs
- Spring provides pre-packaged AOP services:
  - Declarative Transaction Management
  - Security
  - Logging
- You can write custom AOP services for:
  - Auditing
  - Caching
  - Custom security
- Decomposes a system into concerns, instead of objects.

Conceitos de AOP

- Implementação tradicional com todos estes interesses

```
//...
public class AccountService {
 public void updateAccount(Ccount acount) throws Exception {
        SecurityManager.requireUserRole("admin");

        TransactionManager.beginTransaction();
        try {
            //...código de negócio
            TransactionManager.commit();
            log.info("acount updated");

        } catch (Exception e) {
            TransactionManager.rollback();
            log.warning("exception throw", e);
            throw e;
        }
    }
///...
```

Segurança

Transações

Logging

Conceitos de AOP

- Esses interesses são chamados de interesses ortogonais (*Cross cutting Concerns*)
- Sua implementação atravessa a implementação de toda uma classe ou até mesmo várias classes (como logging)
- Com AOP separamos esses interesses e definimo-os como um *advice*
  - Before Advice
  - After Advice
  - Around Advice
  - Throws Advice
- E baseado em certos critérios (*Pointcuts*), estes são inseridos na aplicação (*Weaver*)

Tipos de AOP

- AOP Estático
  - Aspectos são tipicamente introduzidos ao byte code em tempo de compilação ou através de classloaders customizados em tempo de execução
    - AspectJ (byte code)
    - JBoss AOP, AspectWerkz (classloader)
- AOP Dinâmico
  - Cria *proxies* para todos os objetos interessados
  - Leve perda de performance
  - Fácil de configurar
    - Spring

AOP (Aspect-Oriented Programming)

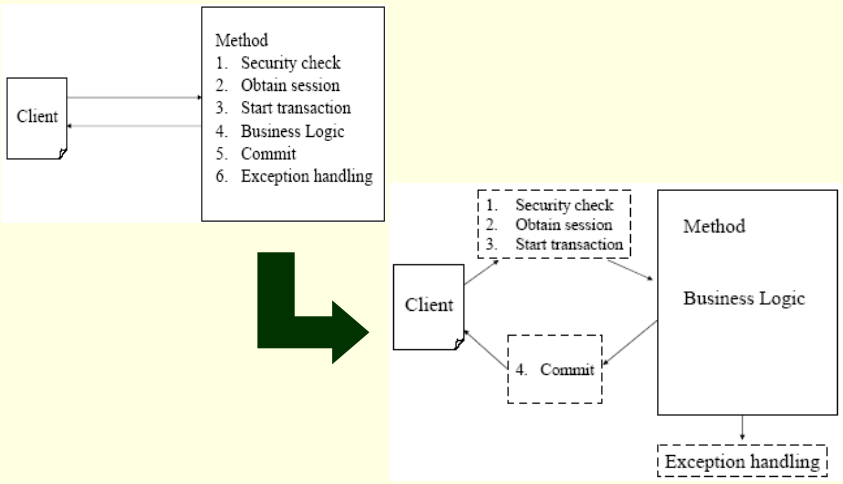
- Deals with "**aspects**" that **cross-cut** across the code and can be difficult or impossible to modularize with OOP
- The most common example given is **logging**
  - Code for doing logging typically must be scattered all over a system
  - With AOP, you can declare, for example, that a system should write a log record at the beginning and end of all method invocations.
- In Spring, AOP
  - uses Dynamic AOP Proxy objects to provide cross-cutting services
  - Reusable components
  - Aopalliance support today
  - AspectJ support in Spring 1.1

Spring

Spring AOP

- Framework that builds on the aopalliance interfaces.
- Aspects are coded with pure Java code. You do not need to learn pointcut query languages that are available in other AOP implementations.
- Spring aspects can be configured using its own IoC container.
  - Objects obtained from the IoC container can be transparently advised based on configuration
- Spring AOP has built in aspects such as providing transaction management, performance monitoring and more for your beans

Spring AOP



Spring

Spring AOP

- Complements OO programming
- Core business concerns vs. Crosscutting enterprise conc
- Components of AOP
  - **Advice** – This is the code that is applied to, or cross-cuts, your existing object model. Advice code is what modifies the behavior or properties of an existing object.
  - Advice is the block of code that runs based on the pointcut definition. Action taken by an aspect at a particular join point. Many AOP frameworks, including Spring, model an advice as an interceptor, maintaining a chain of interceptors "around" the join point. The actual decorator of functionality, for example Logging of all Http requests with a complete list of request parameters
    - Types of Advice
      - **before advice, which executes before joinpoint**
      - **after advice, which executes after joinpoint**
      - **around advice, which executes around joinpoint**

Spring

Spring AOP

- **Pointcut** – define the points in your model where advice will be applied. For example, pointcuts define where in a class code should be introduced or which methods should be intercepted before they are executed. join point queries where advice executes. A collection of Joint Points. By creating pointcuts, you gain fine-grained control over how you apply advice to the components
  - Declaratively stating that all Http request matching a given pattern should be intercepted
- By creating pointcuts, you gain fine-grained control over how you apply advice to the components
  - Example
    - A typical joinpoint is a method invocation.
    - A typical pointcut is a collection of all method invocations in a particular class
- Pointcuts can be composed in complex relationships to further constrain when advice is executed

Spring

Spring AOP

- **Join point** – well-defined points in the program flow. A point during the execution of a program, such as the execution of a method or the handling of an exception. Examples of Jointpoint's:
  - Method invocation
  - Class initialization
  - Object initialization
- Candidates for advice interception
  - Common validations for all Http requests
  - Transaction properties for all service method invocations

Spring

Spring AOP

- **Aspects** – unit of modularity for crosscutting concerns. A modularization of a concern that cuts across multiple objects. Invocations that should be handled similarly.
- An aspect is the combination of advice and pointcuts into functional units in much the same way that OOP uses classes to package fields and methods into cohesive units.
  - For example, you might have a logging aspect that contains advice and pointcuts for applying logging code to all setter and getter methods on objects.
- **Weaving** – Process of actually inserting aspects/advice (crosscutting concerns) into the application code (core concerns) at the appropriate point
  - Types of Weaving
    - **Compile time weaving**
    - **Runtime weaving**

Spring

Spring AOP – Example

- Aspects can be used as an alternative to existing technologies such as EJB. Ex: declarative transaction management, declarative security, profiling, logging, etc.
- Aspects can be added or removed as needed without changing your code.
- **Target -** An object whose execution flow is modified by some AOP process. They are sometimes called advised object
- **Advice example**

```
package swe645.ioc;
import java.lang.reflect.Method;
import org.springframework.aop.*;
public class MyAdvice implements MethodBeforeAdvice{
  public void before(Method m, Object[] args, Object target) {
    System.out.println("Before Advice");
  }
}
```

Spring AOP – Pointcut

- Set of joinpoints specifying when an advice should fire

```
public interface Pointcut {
    ClassFilter getClassFilter();
    MethodMatcher getMethodMatcher();
}

public interface ClassFilter {
    boolean matches(Class clazz);
}

public interface MethodMatcher {
    boolean matches(Method m, Class targetClass);
    boolean matches(Method m, Class targetClass, Object[] args);
    boolean isRuntime();
}
```

Restricts the pointcut to a given set of target classes

Static pointcuts don't use the method arguments

Spring AOP – Pointcut implementation

- Regexp

```
<bean id="gettersAndSettersPointcut"
    class="org.springframework.aop.support.RegexpMethodPointcut">
    <property name="patterns">
        <list>
            <value>.*\.get.*</value>
            <value>.*\.set.*</value>
        </list>
    </property>
</bean>
```

Match a Perl5 regexp to a fully qualified method name

Spring AOP – Advice

- Action taken at a particular joinpoint

```
public interface MethodInterceptor extends Interceptor {
    Object invoke(MethodInvocation invocation) throws Throwable;
}
```

Spring implements an advice with an ***interceptor chain*** around the jointpoint

- Example

```
public class DebugInterceptor implements MethodInterceptor {

    public Object invoke(MethodInvocation invocation)
        throws Throwable {
        System.out.println(">> " + invocation); // before
        Object rval = invocation.proceed();
        System.out.println("<< Invocation returned"); // after
        return rval;
    }
}
```


Spring AOP – Advice managed aspects

- Around (the previous example)
  - MethodInterceptor – add behaviour and continue explitly to the next advice in the chain
- Before
  - MethodBeforeAdvice
- After
  - AfterReturningAdvice
- After throwing exception
  - ThrowsAdvice – specify per exception type or generalise

Spring AOP – Advisors

- PointcutAdvisor = Pointcut + Advice
- Each built-in advice has na advisor
  - Example

```
<bean id="gettersAndSettersAdvisor"
    class="...aop.support.RegexpMethodPointcutAroundAdvisor">
    <property name="interceptor">
        <ref local="interceptorBean"/>
    </property>
    <property name="patterns">
        <list>
            <value>.*\.get.*</value>
            <value>.*\.set.*</value>
        </list>
    </property>
</bean>
```


Spring AOP – Example

- Example (cont.)

```
<bean id="myAdvisor"
  class="o.spf.aop.support.RegexpMethodPointcutAdvisor">
  <property name="advice" ref ="myAdvice"/>
  <property name="pattern">
     <value>.*</value>
   </property>
</bean>
```

Spring AOP – Example

```
<bean id="AOPTest"
class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="proxyInterfaces">
    <value>swe645.ioc.WeatherService</value>
  </property>
  <property name="target" ref ="weatherService"/>
  <property name="interceptorNames">
    <list>
      <value> myAdvisor1</value>
      <value> myAdvisor2</value>
    </list>
  </property>
</bean>
```


Spring AOP – Example

```
ApplicationContext ctx = new
ClassPathXmlApplicationContext("ioc/applicationContext.xml");

  WeatherService ws = (WeatherService)ctx.getBean("AOPTest");

  Double high = ws.getHistoricalHigh(new
                  GregorianCalendar(2004, 0, 1).getTime());
  System.out.println("High was test: " + high);
```

Declarative Spring Transactions

```
<bean id="transactionManager" class=
    "org.springframework.orm.hibernate.HibernateTransactionManager">
  <property name="sessionFactory">
    <ref bean="sessionFactory"/>
  </property>
</bean>

<bean id="studentService" class=
"org.springframework.transaction.interceptor.
  ➥ TransactionProxyFactoryBean">

  <property name="target">
    <ref bean="studentServiceTarget"/>
  </property>

  <property name="transactionAttributes">
    <props>
      <prop key="registerForCourse">
          PROPAGATION_REQUIRES_NEW,ISOLATION_DEFAULT
      </prop>
    </props>
  </property>

  <property name="transactionManager">
      <ref bean="transactionManager"/>
  </property>
```

Declare transaction manager

Apply transactions

Declare transaction

Inject transaction

Service Abstraction Layers

Spring provides abstraction for:

- Transaction Management
  - JTA, JDBC, others
- Data Access
  - JDBC, Hibernate, JDO, TopLink, iBatis
- Email
- Remoting
  - EJB, Web Services, RMI, Hessian/Burlap

Service Abstraction Layers

Benefits:

- No implicit contracts with JNDI, etc.
- Insulates you from the underlying APIs
- Greater reusability
- Spring abstractions always consist of interfaces
- This makes testing simpler
- For data access, Spring uses a generic transaction infrastructure and DAO exception hierarchy that is common across all supported platforms

Spring

Spring AOP

- Supports the following advices:
  - method before
  - method after returning
  - throws advice
  - around advice (uses AOPAlliance MethodInterceptor directly)
- Spring allows you to chain together interceptors and advice with precedence.
- Aspects are weaved together at runtime. AspectJ uses compile time weaving.
- Spring AOP also includes advisors that contain advice and pointcut filtering.
- ProxyFactoryBean – sources AOP proxies from a Spring BeanFactory
- IoC + AOP is a great combination that is non-invasive

Spring AOP Around Advice Example

```
public class PerformanceMonitorDetailInterceptor implements MethodInterceptor
{
   protected final Log logger = LogFactory.getLog(getClass());

   public Object invoke(MethodInvocation invocation) throws Throwable {
       String name = invocation.getMethod().getDeclaringClass().getName()
                         + "." + invocation.getMethod().getName();

       StopWatch sw = new StopWatch(name);
       sw.start(name);
       Object rval = invocation.proceed();

       sw.stop();
       logger.info(sw.prettyPrint());
       return rval;
   }
}
```


Spring AOP Advice (Cont')

- Advisor references the advice and the pointcut

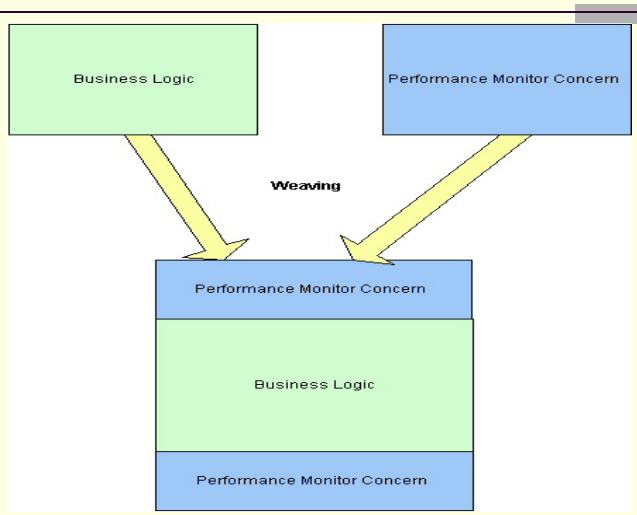
```
<bean id="perfMonInterceptor"
   class=
   "com.meagle.service.interceptor.PerformanceMonitorDetailInterceptor"/>

<bean id="performanceAdvisor"
   class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
   <property name="advice">
        <ref local="perfMonInterceptor"/>
    </property>

    <property name="patterns">
        <list>
            <value>.*find.*</value>
            <value>.*save.*</value>
            <value>.*update.*</value>
        </list>
    </property>
</bean>
```

Spring

AOP Weaving



Spring

Wiring Beans Together with Spring

Wiring your Persistence Layer

```
<bean id="mySessionFactory" class=
"org.springframework.orm.hibernate.LocalSessionFactoryBean">
    <property name="mappingResources">
      <list>
        <value>com/matrix/bo/Order.hbm.xml</value>
        <value>com/matrix/bo/OrderLineItem.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <props>
        <prop key="hibernate.dialect">
                net.sf.hibernate.dialect.DB2Dialect
        </prop>
        <prop key="hibernate.default_schema">DB2ADMIN</prop>
        <prop key="hibernate.show_sql">false</prop>
        <prop key="hibernate.proxool.xml">
                /WEB-INF/proxool.xml</prop>
        <prop key="hibernate.proxool.pool_alias">spring</prop>
      </props>
    </property>
</bean>
```


Wiring your Transaction Management

Four transaction managers available

- DataSourceTransactionManager
- HibernateTransactionManager
- JdoTransactionManager
- JtaTransactionManager

```
<bean id="myTransactionManager"
      class="org.springframework.orm.hibernate
             .HibernateTransactionManager">

   <property name="sessionFactory">
       <ref local="mySessionFactory"/>
   </property>
</bean>
```

Spring

Wiring a Service Object

```
<bean id="orderService" class="org.springframework.transaction.
                        interceptor.TransactionProxyFactoryBean">

  <property name="transactionManager">
      <ref local="myTransactionManager"/>
  </property>
  <property name="target"><ref local="orderTarget"/></property>
  <property name="transactionAttributes">
     <props>
        <prop key="find*">
              PROPAGATION_REQUIRED,readOnly,-OrderException
        </prop>
        <prop key="save*">
              PROPAGATION_REQUIRED,-OrderMinimumAmountException
        </prop>
        <prop key="update*">
              PROPAGATION_REQUIRED,-OrderException
        </prop>
     </props>
  </property>
</bean>
```


Spring

Defining a Target to Proxy

```
public class OrderServiceSpringImpl implements
  IOrderService {

  private IOrderDAO orderDAO;

  // service methods…

  public void setOrderDAO(IOrderDAO orderDAO) {
      this.orderDAO = orderDAO;
  }
}
```

Wiring a Service Object (cont')

```
<bean id="orderTarget"
   class="com.meagle.service.spring.OrderServiceSpringImpl">
   <property name="orderDAO">
        <ref local="orderDAO"/>
   </property>
</bean>

<bean id="orderDAO"
   class="com.meagle.dao.hibernate.OrderHibernateDAO">
   <property name="sessionFactory">
        <ref local="mySessionFactory"/>
   </property>
</bean>
```


Wiring a Service Object (cont')